R V College of Engineering, Bengaluru – 560059

(Autonomous Institution Affiliated to VTU, Belgaum)

2023 - 2024

Biodiversity Analysis using Probabilistic Data Structures

EXPERIENTIAL LEARNING REPORT

Data Structures and Applications

IS233AI

Submitted By –

Arya Hariharan – 1RV22CS029

Gayatri K – 1RV22CS057

Faculty Mentor - Dr. Girish Rao Salanke

Assistant Professor, Dept. of Computer Science and Engineering, RVCE

TABLE OF CONTENTS

- Introduction 1.

1. Biodiversity and Indian Ecosystems 2.
   1.1. Importance of Biodiversity Analysis
   1.2. Dataset
2. Objectives and Methodology 4.
   2.1. Primary Objectives
   2.2. Methodology
3. Probabilistic Data Structures 5.
   3.1. Hashing
   3.2. Bloom Filters
   3.3. Heavy Keeper
   3.4. HyperLogLog
4. Real Life Case Studies 15.
5. Results 17.

- Conclusion 19.
- References 20.

INTRODUCTION

Probabilistic data structures (PDS) are a unique type of data structure that differs from traditional deterministic data structures. Unlike deterministic data structures like arrays and trees, which always provide exact answers, PDS use randomization to provide approximate answers with a guaranteed level of accuracy. This trade-off between accuracy and efficiency allows PDS to handle large datasets effectively, where traditional methods would be too slow and require too much memory.

PDS rely on probabilistic algorithms and techniques, such as hashing and sketching. These algorithms use randomness in a controlled way to organize and retrieve data. This enables PDS to process large datasets with less memory and faster than traditional methods. This makes them especially useful for real-time data analysis and applications where getting an answer quickly is more important than getting an exact answer. Network and server request analysis is one of the most important applications of PDS. A server of a major social media or online service like YouTube can receive millions of requests per second. While databases can store and handle these million of requests, the amount of space required is huge and the time required to parse through theses requests and make certain conclusions is considerably large. PDS can do the same tasks at a fraction of time and space, with a space trade-off in terms of accuracy, which is generally insignificant considering the amount of data being processed.

Thus, PDS represent a new way of designing data structures that prioritizes speed and space efficiency over perfect accuracy. Their ability to handle large datasets effectively makes them essential tools for many different areas, such as streaming data analysis, network monitoring, and machine learning, where getting quick insights and using resources efficiently are crucial. In this report, PDS like HyperLogLog and Count Min-Sketch with min heaps have been used to analyse biodiversity in Indian forests and draw conclusions relating to the number of unique species, families, genera, etc. present in Indian forests, along with their numbers. The report shall expand on the method of implementation and the probabilistic algorithms applied to optimize the use of these data structures.

1. BIODIVERSITY AND INDIAN ECOSYSTEMS

Biodiversity signifies the multi-faceted nature of life on Earth, encompassing variation at the genetic (within-species), species (richness and abundance), and ecosystem (functional processes and community interactions) levels[1]. This intricate tapestry, woven through billions of years of evolution, forms the very foundation for the health and functioning of our biosphere.

India, despite occupying only a small fraction of the world's landmass, boasts exceptional biodiversity richness. This manifests in the vast array of plant and animal species, with a significant proportion being endemic, meaning they exist nowhere else on Earth. The country is further distinguished by the presence of several biodiversity hotspots, highlighting the importance of conservation efforts to protect this irreplaceable natural heritage.

1.1. Importance of Biodiversity Analysis

Biodiversity analysis, encompassing the quantification and characterization of biological diversity at various levels (genetic, species, and ecosystem), plays a pivotal role in understanding and managing our planet's natural world. This multifaceted approach holds significant scientific and practical importance:

- **Understanding Ecosystem Functioning:** By analysing species composition, richness, and functional traits, scientists gain insights into the intricate web of interactions that govern ecosystem functioning. This knowledge informs conservation strategies by identifying keystone species, vulnerable populations, and potential disruptions to ecosystem services such as nutrient cycling and pollination[2].
- **Assessing Conservation Needs:** Biodiversity analysis serves as a crucial tool for prioritizing conservation efforts. Through techniques like gap analysis and species distribution modelling, researchers can identify areas of high biodiversity value and potential threats, thereby guiding the allocation of resources and the design of effective conservation plans[3].
- **Monitoring Biodiversity Change:** Continued monitoring of biodiversity through standardized and long-term analysis enables scientists to detect trends in species abundance, distribution, and community composition. This information serves as an early warning system for potential ecosystem degradation and informs adaptive management strategies[4].

In conclusion, biodiversity analysis is an indispensable tool for comprehending the complex tapestry of life on Earth and guiding efforts to conserve its rich tapestry for generations to come.

1.2. Dataset

As highlighted, biodiversity analysis is extremely important. Thus, the primary objective of this project is to analyse the biodiversity of Indian forests to identify number of unique species, number of species belonging to a phylum, kingdom, etc.

The Indian Biodiversity Portal has, for public use and good, published a dataset comprising of 89,162 records of all observation data of biodiversity found in Indian forests, with records comprising of the species name, family, genus, phylum, etc. The dataset is available in TSV format for download. For the purpose of biodiversity analysis in this project, this dataset shall be utilised for biodiversity analysis. The methodology of analysis is explained in the next section.

The primary expected outcomes involve developing a system through which the cardinality and frequency of the elements of the dataset can be understood, along with the provision to perform a membership query on the dataset. The implementation can be extended to include the creation a biodiversity dashboard using Flask to create a web application which reflects these outcomes.

2. OBJECTIVES AND METHODOLOGY

As highlighted in the previous section, biodiversity analysis is crucial in understanding our ecosystems and how they work. This project aims at conducting biodiversity analysis of the dataset mentioned in the previous section.

2.1. Primary Objectives

- Analyse a dataset comprising of all observation data from the Indian Biodiversity Portal (with about 89,162 records).
- Find the top-k largest genera and families present in Indian forests.
- Implement a search feature to allow the user to check for the presence of a particular species in Indian forests.
- Obtain figures related to number of distinct kingdoms, phylum, classes, orders, families, genera and species of living things present in Indian forests.
- Apply space and time-efficient methods of implementing the same.

2.2. Methodology

Since the primary objectives also includes implementing a space and time-efficient solution, traditional methods of database analysis may not satisfy one or both of these requirements. Hence, the methodology chosen to implement this analysis is by using probabilistic data structures, which are implemented in Python. Thus, the stated objectives with the method of execution are –

- Analyse a dataset comprising of all observation data from the Indian Biodiversity Portal (with about 89,162 records), utilising the Pandas library and DataFrames for initial processing.
- Find the top-k largest genera and families present in Indian forests using Count-Min-Sketch probabilistic data structure and min heap, essentially implementing a heavy keeper algorithm.
- Implement a search feature, supported by Bloom filters, to allow the user to check for the presence of a particular species in Indian forests.
- Obtain figures related to number of distinct kingdoms, phylum, classes, orders, families, genera and species of living things present in Indian forests using HyperLogLog algorithm.

The brief methodology or steps to be followed are –

- Download the dataset from the portal and perform initial cleaning and filtering, either using the Pandas library and Jupyter Notebook or by using Pandas directly within the Python code for each data structure. For this project, initial preprocessing using Jupyter Notebook was the preferred method.
- Implement modules for each data structure, implementing necessary add and query functions.
- Import these modules and the dataset and draw relevant conclusions as per user requirement.

3. PROBABILISTIC DATA STRUCTURES

Probabilistic data structures constitute a unique paradigm within the field of computer science, deviating from the deterministic approach by offering approximate answers to queries about large datasets with guaranteed error bounds. This trade-off between absolute accuracy and efficiency empowers these data structures to excel in scenarios where traditional data structures would be hindered by resource constraints. Probabilistic data structures leverage the power of randomization through techniques like hashing and probabilistic algorithms, allowing them to represent and manipulate data in a space-efficient manner while facilitating rapid processing of vast information streams. This makes them particularly valuable in domains such as real-time analytics, network monitoring, and machine learning, where timely insights and efficient resource utilization are paramount[5][6]. Here, the data structures are used to analyse biodiversity data.

3.1. Hashing

An important concept that is applied in almost all probabilistic data structures, including the ones being used in this project, is hashing. Hashing, a cornerstone of computer science, refers to the process of transforming arbitrary data into fixed-length values, called hash values. This transformation, often performed by a hash function, aims to map similar data elements to the same or similar hash values, while significantly different data maps to distinct values. This characteristic enables efficient data organization and retrieval in various applications, particularly in situations involving large datasets. Notably, hashing plays a crucial role in various data structures like hash tables, where it facilitates fast average-case lookups, insertions, and deletions[7]. Additionally, hashing underpins cryptographic techniques like message authentication codes and digital signatures, where it ensures data integrity and authenticity[8].

Hashing allows for the creation of a unique fingerprint for each value. These are different types of hash functions but the most commonly used are based on number theory and modular arithmetic. In the most fundamental applications, hashing involves converting an input into an output value which acts as an index for where that input element will be stored within a data structure, like an array. In essence, a hash function H is a mathematical function which projects a value from a set with many (or even an infinite number of) members to a value from a set with a fixed number of (fewer) members. Hash functions are not reversible.

Python offers in-built support for different types of hash functions based on modular arithmetic through the **hashlib** library. For the implementation of the probabilistic data structures used in this project, the same has been used.

3.2. Bloom Filters

The first type of probabilistic data structure used in the Bloom filter. Bloom filters, introduced by Burton H. Bloom in 1970[9], are space-efficient probabilistic data structures designed for membership testing in sets. Unlike traditional deterministic data structures, Bloom filters utilize bit arrays and a battery of hash functions to achieve an efficient trade-off between space complexity and query accuracy. Elements are added to the filter by applying multiple hash functions to their key and setting the corresponding bits in the array. Membership queries involve applying the same hash functions to the key and checking if all the corresponding bits are set. While Bloom filters offer fast membership testing and minimal space requirements, they are probabilistic in nature and can yield false positives, meaning the filter might claim membership for elements not present in the set. However, the probability of false positives can be controlled by adjusting the size of the bit array and the number of hash functions used[10]. This inherent trade-off between accuracy and efficiency makes Bloom filters particularly valuable in applications where fast approximate membership queries are crucial[11], and for this project, it has been utilised for implementing the search feature.

Expanding on the above introduction to Bloom filters, consider a set A with *n* number of elements which are to be added to a data structure to be used for membership query analysis.

$$A = \{a_1, a_2, \dots, a_n\}$$

To create a Bloom filter, allocate a vector *v* of *m* bits, initially all set to 0. Also create *k* independent hash functions, say -

$$h_1, h_2, \dots, h_k$$

each with a range from 0 to m.

To insert an element, apply all *k* hash functions on the element. Each element returns a value ranging between 0 and m. For all the values returned from the hash function, treat them as indices and set all those indices within the vector/array to 1, if not already one. Repeat for all elements. To check for an element, apply the *k* hash functions on the element can check the values of the array at the indices returned as outputs from the hash function. If any one of those values within the array at the specified indices is 0, the element with 100% surety has not been entered into the dataset and thus is not present within the dataset.

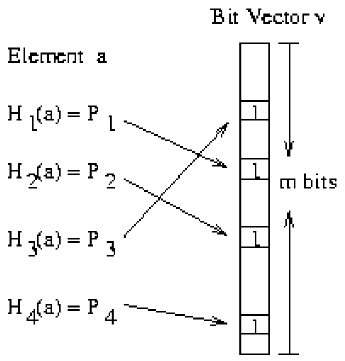


Fig : Adding element into the vector/array

However, if all of the values of the array are 1, we cannot say with 100% surety that the tested element is present in the dataset. This is due to the possibility of collision, where two elements have the exact same output values for the *k* hash functions. Thus, even if an element is not present, a false positive can be generated due to the collision, where it seems as if the element is actually present. This is the “error” that is associated with this probabilistic data structure. However, by applying mathematical techniques, it is possible to reduce this error rate and set it to a desired value by optimising the value of *k* and *m*.

We assume that our hash functions work uniformly, i.e., for any given input they are equally likely to select any of the m lots in the array. Thus, the probability that a bit is set to 1 for one hash function is given by first equation. Probability that a bit is 0 for one hash function is given by the second equation.

$$p_i = \frac{1}{m}$$

$$p'_i = 1 - p_i = 1 - \frac{1}{m}$$

Probability that a bit is 0 for *k* hash functions is given by essentially the above expression multiplied *k* times. Based on these three equations, we can conclude that the probability that a bit is 0 for *k* hash functions after *n* insertions is given by the second equation -

$$(1 - \frac{1}{m}) \cdot (1 - \frac{1}{m}) \cdot .... \cdot (1 - \frac{1}{m}) = (1 - \frac{1}{m})^k$$

$$(p'_i)^{nk} = \left(1 - \frac{1}{m}\right)^{nk}$$

Assume now a membership query is to be done. The probability that all indices returned by the *k* hash functions have values 1 in the array is given by -

$$\left[1 - (p'_i)^{nk}\right]^k = \left[1 - \left(1 - \frac{1}{m}\right)^{nk}\right]^k$$

For larger values of m, this can be approximated as -

$$\lim_{m\to\infty} (p'_i)^m = \lim_{m\to\infty} \left(1 - \frac{1}{m}\right)^m = \frac{1}{e}$$

$$\implies \lim_{m\to\infty} \left[1 - (p'_i)^{nk}\right]^k = \lim_{m\to\infty} \left[1 - ((p'_i)^m)^{nk/m}\right]^k = \left[1 - \exp\left\{\frac{-nk}{m}\right\}\right]^k = \epsilon(k)$$

This probability essentially represents the possibility of error, which is required to minimized. To do so, we need to differentiate this function and equate to 0. On differentiating, we obtain the optimized value of *k* for minimum possibility of error -

$$\epsilon(k) = \left(1 - \exp\left\{\frac{-nk}{m}\right\}\right)^k \implies f(k) = \log\epsilon(k) = k\log\left(1 - \exp\left\{\frac{-nk}{m}\right\}\right)$$

$$\frac{\mathrm{d}f}{\mathrm{d}k} = \frac{\mathrm{d}}{\mathrm{d}k}k\log\left(1 - \exp\left\{\frac{-nk}{m}\right\}\right) = \log\left(1 - \exp\left\{\frac{-nk}{m}\right\}\right) + k\frac{\mathrm{d}}{\mathrm{d}k}\log\left(1 - \exp\left\{\frac{-nk}{m}\right\}\right)$$

$$\text{Let } \ \alpha(k) = \exp\left\{\frac{-nk}{m}\right\} \implies \frac{\mathrm{d}f}{\mathrm{d}k} = \log\left(1 - \alpha(k)\right) + \frac{nk}{m}\cdot\frac{\alpha(k)}{1-\alpha(k)} = 0$$

$$\implies \log\left(1 - \alpha(k)\right) - \log\left(\alpha(k)\right)\cdot\frac{\alpha(k)}{1-\alpha(k)} = 0 \implies \frac{\left(1-\alpha(k)\right)\log\left(1-\alpha(k)\right) - \log\left(\alpha(k)\right)\alpha(k)}{1-\alpha(k)} = 0$$

$$\implies 1 - \alpha(k) = \alpha(k) \implies \alpha(k) = \frac{1}{2} = \exp\left\{\frac{-nk}{m}\right\}$$

$$\implies k = \frac{m}{n}\log 2$$

Now, the error rate using this value can be represented as -

$$\epsilon(k) = \left[1 - \exp\left\{\frac{-nk}{m}\right\}\right]^k \implies \bar{\epsilon} = \epsilon\left(\frac{m}{n}\log 2\right) = \left[1 - \exp\left\{\frac{-nm}{mn}\cdot\log 2\right\}\right]^{\frac{m}{n}\log 2} = \left(\frac{1}{2}\right)^{\frac{m}{n}\log 2}$$

Using this minimum error rate, we can find the value of *m*.

$$\bar{\epsilon} = \left(\frac{1}{2}\right)^{\frac{m}{n}\log 2} \implies \log\bar{\epsilon} = \frac{m}{n}\log 2\cdot\log\frac{1}{2}$$

$$\implies m^* = -\frac{n\log\bar{\epsilon}}{(\log 2)^2}$$

Thus, through this, we can find the optimized values of *m* and *k* if we wish to implement Bloom filter with a particular error rate. For example, if we wish to create a Bloom filter with an error rate of 1% and check for membership within 200,000 entries, the values of *m* and *k* would be -

$$m^* = -\frac{n\log\bar{\epsilon}}{(\log 2)^2} = -\frac{200000\log 0.01}{(\log 2)^2} = 1917011$$

$$k^* = \frac{m^*}{n}\log 2 = \frac{1917011}{200000}\log 2 = 6.64 \approx 7$$

This data structure has a both constant space and time complexity as it depends on the value of *m* and *k* respectively, which always remains a constant.

3.3. Heavy Keeper

The heavy keeper is an algorithm which utilises the Count Min Sketch probabilistic data structure and a min heap to find the top-k elephant flows in a dataset.

The count-min sketch (CMS) is a probabilistic data structure designed to estimate the frequency of items in a data stream. Unlike traditional data structures, which require linear space in the number of items, the CMS leverages hashing and probabilistic techniques to achieve sub-linear space complexity. This efficiency comes at the cost of accuracy, as the CMS utilizes multiple hash functions to map items to a compact data structure, potentially causing collisions and leading to slight overestimations of individual item frequencies[12]. However, the CMS guarantees an upper bound on the error for any item, making it a valuable tool for applications where space efficiency and real-time processing are crucial.

Expanding on the above introduction to count-min sketch, consider a set A with *n* number of elements which are to be added to a data structure so as to calculate the frequency of elements in the set.

$$A = \{a_1, a_2, \dots, a_n\}$$

To create a count-min sketch, allocate a two-dimensional array of *k* x *m* bits, initially all set to 0. Also create *k* independent hash functions, say -

$$h_1, h_2, \dots, h_k$$

each with a range from 0 to m.

Consider a count-min sketch with *m* = 5 and *k* = 2. Thus, the initial state of the array will be –

0	0	0	0	0
0	0	0	0	0

Assume the element 2 is being added. The output of the two hash functions will be say 3 for the first hash function and 1 for the second. Assume now the element 1 is being added. The output of the first function is 3 again and the output of the second is 0. On inserting two elements, the array now looks like -

0	0	0	2	0
1	1	0	0	0

Now, suppose we want to check for the frequency of the element 2. Once again, we hash it against the two functions, which return the indices of 3 and 1 for the first and second function respectively. The value at these positions within the array is 1 and 2, out of which the minimum is 1. Count-min sketch will always return the minimum value, which in this case is 1. Due to this being a relatively simple example, we get the right output. However, in general, the minimum value returned can also be higher than the actual frequency due to its increase by the addition of other functions with the same hash value outputs. Thus, count-min sketch has a tendency of overestimation.

We can limit this tendency by using probabilistic algorithms. We know that the estimated frequency can be equal to or greater than the actual frequency as given in Eq(1).

$$\hat{f}_x \geq f_x$$

We can ensure that the probability that the difference is lesser than a set value, as per requirement will be greater than or equal to a factor equal to or lesser than one, which again can be set as per user requirement.

$$P(\hat{f}_x - f_x \leq \epsilon n) \geq 1 - \delta$$

We know that

$$\hat{f}_x = \min_{i \in \{0,1,\cdots,k\}} \alpha_{i,h_i(x)}$$

Let’s define a random variable representing the overcount.

$$Z_i = \alpha_{i,h_i(x)} - f_x$$

Let’s also define a random variable whose value depends on whether collision occurs for a particular hash function or not. This is an indicator random variable.

$$X_{i,y} = \begin{cases} 1 & \text{y collides with x in the ith row} \\ 0 & \text{otherwise} \end{cases} = \begin{cases} 1 & h_i(y) = h_i(x) \\ 0 & \text{otherwise} \end{cases}$$

The random variable can be calculated as -

$$Z_i = \sum_{y \neq x} f_y X_{i,y}$$

The expected value of the random variable is -

$$\begin{aligned} Z_i = \sum_{y\neq x} f_y X_{i,y} \implies \mathbb{E}\left[Z_i\right] &= \mathbb{E}\left[\sum_{y\neq x} f_y X_{i,y}\right] \\ &= \sum_{y\neq x} f_y \mathbb{E}\left[X_{i,y}\right] \quad \cdots \text{ by linearity of expectation} \\ &= \sum_{y\neq x} f_y P(h_i(y) = h_i(x)) \quad \cdots \text{ by definition of indicator variable} \\ &= \sum_{y\neq x} f_y \cdot \frac{1}{m} \quad \cdots \text{ by uniformity of the hashing functions} \\ &= \frac{1}{m}\sum_{y\neq x} f_y \leq \frac{n}{m} \end{aligned}$$

Using Markov's inequality, we can arrive at the following conclusions –

$$P\left(Z_i \geq a\mathbb{E}\left[Z_i\right]\right) \leq \frac{1}{b}$$

$$P\left(Z_i \geq a\frac{n}{m}\right) \leq P\left(Z_i \geq a\mathbb{E}\left[Z_i\right]\right) \leq \frac{1}{b}$$

$$\implies P\left(Z_i \geq a\frac{n}{m}\right) \leq \frac{1}{b}$$

Applying the followin g transformation –

$$a = m\epsilon \implies P\left(Z_i \geq \epsilon n\right) \leq \frac{1}{m\epsilon}$$

This implies that –

$$P\left(Z_i \geq \epsilon n\right) \leq \frac{1}{e} \implies P\left(\alpha_{i,h_i(x)} - f_x \geq \epsilon n\right) \leq \frac{1}{e} \implies P\left(\alpha_{i,h_i(x)} \geq f_x + \epsilon n\right) \leq \frac{1}{e}$$

$$P\left(\alpha_{i,h_i(x)} \geq f_x + \epsilon n\right) \leq \frac{1}{e} \implies P\left(\left(\forall i \in \{1,2,\cdots,k\}\right) \alpha_{i,h_i(x)} \geq f_x + \epsilon n\right) \leq \left(\frac{1}{e}\right)^k$$

Thus, we can conlude that -

$$\therefore P\left(\min_{i\in\{1,2,\cdots,k\}} \alpha_{i,h_i(x)} \geq f_x + \epsilon n\right) \leq \left(\frac{1}{e}\right)^k \implies P\left(\hat{f}_x \geq f_x + \epsilon n\right) \leq \left(\frac{1}{e}\right)^k$$

Manipulating the above equation –

$$P\left(\hat{f}_x \geq f_x + \epsilon n\right) \leq \left(\frac{1}{e}\right)^k \implies P\left(\hat{f}_x \leq f_x + \epsilon n\right) \geq 1 - \left(\frac{1}{e}\right)^k$$

$$\implies P\left(\hat{f}_x \leq f_x + \epsilon n\right) \geq 1 - \left(\frac{1}{e}\right)^{\log(1/\delta)} = 1 - \delta$$

Based on this, we get the optimized value of *m* and *k* which is given by -

$$m = \frac{e}{\epsilon};\ k = \log\left(\frac{1}{\delta}\right) \implies \text{Space: } O(mk) = O\left(-\epsilon^{-1}\log\delta\right)$$

The above equation also states the space complexity taken. Again, it is constant and so is the time complexity as the number of hash functions does not change.

Once a count-min sketch has been constructed, we create a min heap to assess the top-K elephant flows. This means that the size of the min heap will be K. As is the property of heaps, when deletion takes place, the root node will always be deleted. Due to the structural property of min heap, the lowest element will always be present as the root node. Adding the first K elements from the sketch, the minimum value will be stored in the min heap. On adding another element, the heap will be heapified again and result in a heap of K+1 elements with the root node holding the minimum element. To maintain a K sized heap, the root node will be replaced. This process will be continued until all frequencies have been parsed through and finally, the min heap will contain only the top-K heaviest/high frequency elements. In this way, count-min sketch with a min heap can be used to implement a heavy keeper data structure.

3.4. HyperLogLog

HyperLogLog is a probabilistic data structure designed to estimate the cardinality (number of distinct elements) of a multiset, particularly in scenarios where exact counting is impractical due to memory limitations. Unlike traditional methods that require storing each unique element, HyperLogLog employs a probabilistic approach that utilizes hash functions and bit arrays. Each element in the multiset is hashed, and the resulting bit string is analysed to determine the leading zeros. The estimate of the cardinality is then derived based on the statistical properties of leading zeros across the hashed elements[13]. This approach allows HyperLogLog to maintain a small, fixed memory footprint while offering a guaranteed level of accuracy, typically within a few percent of the true cardinality[14].

The basis for its application is the LogLog algorithm, based on the Flajonet-Martin cardinality law. Modifications in the low led to the SuperLogLog algorithm and finally the HyperLogLog algorithm. The major improvisation on the HyperLogLog algorithm is that it splits the multiset into subsets and estimates their cardinalities, then it uses the harmonic mean to combine them into an estimate for the original cardinality.

4. REAL LIFE CASE STUDIES

These data structures and algorithms are pivotal in computer science, especially for data processing and analytics. Each has its specific use cases, often in the context of large-scale data management, networking, and probabilistic data structures for efficient computation. Here are real-life case studies or applications for each:

4.1. Hashing

Use Case : Secure Password Storage

In the realm of web development and security, hashing plays a crucial role in storing user passwords securely. Companies like Facebook, Google, and Twitter employ hashing algorithms to protect user passwords. When a user creates an account or changes their password, the system hashes the password and stores this hash instead of the actual password. This method ensures that even if data breaches occur, the attackers cannot easily decipher user passwords, as reversing a hash function is computationally infeasible for strong hash functions.

Moreover, to enhance security, salt (a random value) is added to passwords before hashing. This approach prevents attacks using rainbow tables (precomputed tables for reversing cryptographic hash functions). For example, LinkedIn improved its password storage security practices by implementing salting and hashing after a significant breach exposed unsalted password hashes.

4.2. Bloom Filters

Use Case : Web Caching Systems

Bloom filters have a significant application in web caching mechanisms to efficiently determine whether a web page is cached. A real-life application can be observed in the Squid web proxy cache. Squid uses Bloom filters to quickly check if the requested URL is available in the cache. This process significantly reduces the cache lookup time, as it avoids an exhaustive search through the entire cache.

The advantage of using a Bloom filter lies in its space efficiency and the speed of queries. However, it comes at the cost of a small probability of false positives (where the filter mistakenly indicates that a URL is in the cache when it's not), which is a trade-off that systems like Squid are willing to accept for improved overall performance.

4.3. Heavy Keeper

Use Case : Internet Traffic Analysis

Heavy Keeper's utility shines in internet traffic analysis, especially for detecting anomalies and managing network resources. A detailed example involves its use by internet service providers

(ISPs) to monitor and analyze the traffic flow over their networks. By implementing Heavy Keeper, ISPs can identify the most frequently visited websites or the most bandwidth-intensive services. This information is crucial for network planning and ensuring quality of service.

For instance, during the COVID-19 pandemic, ISPs saw a massive shift in internet usage patterns, with a significant increase in video conferencing and streaming services traffic. By using Heavy Keeper, ISPs could adapt to these changes more effectively, allocating resources dynamically and upgrading infrastructure as needed to handle the increased load, thus maintaining service quality and user experience.

4.4. HyperLogLog

Use Case : Real-time Analytics in Social Media Platforms

HyperLogLog is invaluable for real-time analytics, particularly on social media platforms like Twitter or Instagram, where it's used to estimate the number of unique users who have seen a post, hashtag, or story. This estimation is crucial for both the platform and its users, particularly content creators and advertisers, who rely on engagement metrics to gauge the impact of their content.

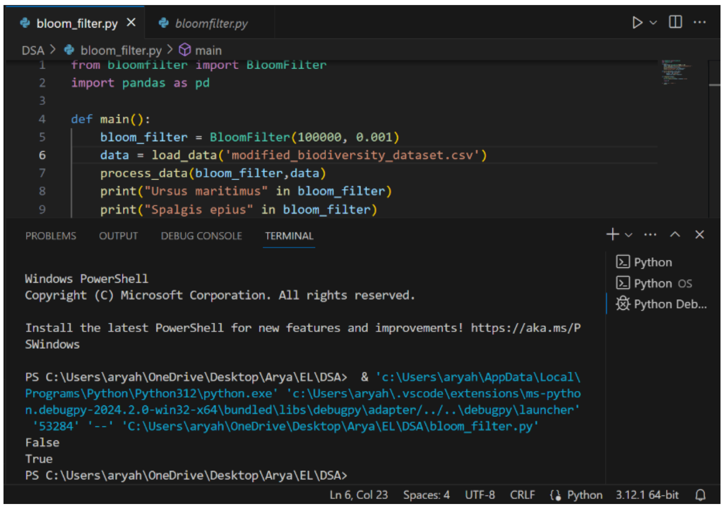
For example, Instagram stories feature might use HyperLogLog to estimate the unique views of a story. Given the platform's vast user base, accurately counting each unique view without duplication in real-time would require substantial memory and processing power. HyperLogLog allows Instagram to provide these metrics with minimal resource usage, enabling content creators to receive immediate feedback on their posts' reach and engagement.

5. RESULTS

The probabilistic data structures mentioned in the previously were implemented using Python. The dataset was then analysed.

Each data structure was initialised with the required error rate and maximum possible number of elements and the results were analysed. Due to the relatively smaller size of the dataset, the same results were drawn using Pandas and Jupyter notebook for the purposes of comparison. The corresponding results were a good match, showing the effectiveness of the use of probabilistic data structures.

Bloom filters were applied to check whether a species was present or not within the dataset. When tried with the species "Urusus maritimus" or the polar bear and with the species "Spalgis epius" or the apefly, the results returned False and True respectively, which is as expected.



```python
from bloomfilter import BloomFilter
import pandas as pd

def main():
    bloom_filter = BloomFilter(100000, 0.001)
    data = load_data('modified_biodiversity_dataset.csv')
    process_data(bloom_filter,data)
    print("Ursus maritimus" in bloom_filter)
    print("Spalgis epius" in bloom_filter)
```

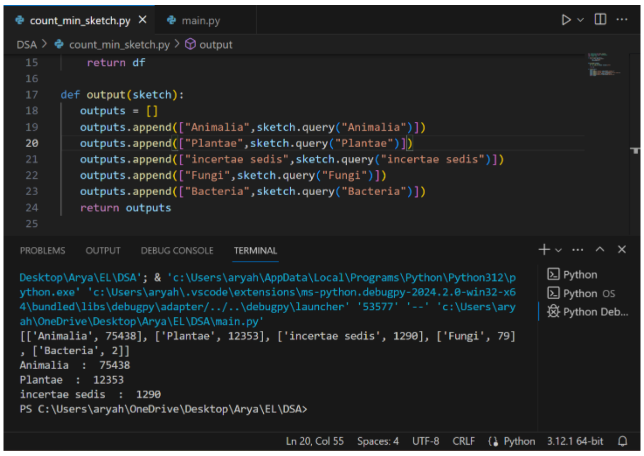
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/P
SWindows

PS C:\Users\aryah\OneDrive\Desktop\Arya\EL\DSA>  & 'c:\Users\aryah\AppData\Local\
Programs\Python\Python312\python.exe' 'c:\Users\aryah\.vscode\extensions\ms-pytho
n.debugpy-2024.2.0-win32-x64\bundled\libs\debugpy\adapter/../..\debugpy\launcher'
 '53284' '--' 'C:\Users\aryah\OneDrive\Desktop\Arya\EL\DSA\bloom_filter.py'
False
True
PS C:\Users\aryah\OneDrive\Desktop\Arya\EL\DSA>
```

Fig : Results of Bloom filter

On implementing a heavy keeper with a min heap of size 3 for kingdoms alone, it successfully returns the top three kingdoms of all the kingdoms present within the dataset based on frequency encountered. The same can be applied for the other features.



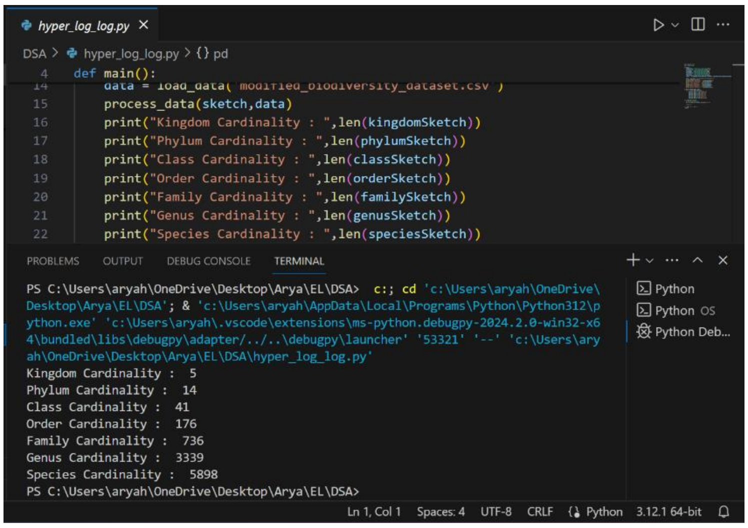
```python
        return df

def output(sketch):
    outputs = []
    outputs.append(["Animalia",sketch.query("Animalia")])
    outputs.append(["Plantae",sketch.query("Plantae")])
    outputs.append(["incertae sedis",sketch.query("incertae sedis")])
    outputs.append(["Fungi",sketch.query("Fungi")])
    outputs.append(["Bacteria",sketch.query("Bacteria")])
    return outputs
```

```
Desktop\Arya\EL\DSA'; & 'c:\Users\aryah\AppData\Local\Programs\Python\Python312\p
ython.exe' 'c:\Users\aryah\.vscode\extensions\ms-python.debugpy-2024.2.0-win32-x6
4\bundled\libs\debugpy\adapter/../..\debugpy\launcher' '53577' '--' 'c:\Users\ary
ah\OneDrive\Desktop\Arya\EL\DSA\main.py'
[['Animalia', 75438], ['Plantae', 12353], ['incertae sedis', 1290], ['Fungi', 79]
, ['Bacteria', 2]]
Animalia  :  75438
Plantae  :  12353
incertae sedis  :  1290
PS C:\Users\aryah\OneDrive\Desktop\Arya\EL\DSA>
```

Fig : Top three kingdoms

On implementing the HyperLogLog, it successfully returns the number of distinct kingdoms, genera, families, species, etc. of animals and plants present in the dataset.



```python
def main():
    data = load_data('modified_biodiversity_dataset.csv')
    process_data(sketch,data)
    print("Kingdom Cardinality : ",len(kingdomSketch))
    print("Phylum Cardinality : ",len(phylumSketch))
    print("Class Cardinality : ",len(classSketch))
    print("Order Cardinality : ",len(orderSketch))
    print("Family Cardinality : ",len(familySketch))
    print("Genus Cardinality : ",len(genusSketch))
    print("Species Cardinality : ",len(speciesSketch))
```

```
PS C:\Users\aryah\OneDrive\Desktop\Arya\EL\DSA>  c:; cd 'c:\Users\aryah\OneDrive\
Desktop\Arya\EL\DSA'; & 'c:\Users\aryah\AppData\Local\Programs\Python\Python312\p
ython.exe' 'c:\Users\aryah\.vscode\extensions\ms-python.debugpy-2024.2.0-win32-x6
4\bundled\libs\debugpy\adapter/../..\debugpy\launcher' '53321' '--' 'c:\Users\ary
ah\OneDrive\Desktop\Arya\EL\DSA\hyper_log_log.py'
Kingdom Cardinality :  5
Phylum Cardinality :  14
Class Cardinality :  41
Order Cardinality :  176
Family Cardinality :  736
Genus Cardinality :  3339
Species Cardinality :  5898
PS C:\Users\aryah\OneDrive\Desktop\Arya\EL\DSA>
```

Ln 1, Col 1   Spaces: 4   UTF-8   CRLF   Python   3.12.1 64-bit

Fig : Cardinality of different features

All the results produced (which can be obtained on running the code present in the Google Drive folder) is in correspondence with the results received on Pandas analysis.

CONCLUSION

In conclusion, probabilistic data structures (PDS) have emerged as a powerful tool for managing and analysing large datasets in diverse fields like bioinformatics, network security, and recommendation systems. Their ability to provide approximate answers with controlled error rates makes them particularly valuable when exact answers are less crucial than efficiency and scalability. As the volume and complexity of data continue to grow, PDS are poised to play an increasingly significant role in extracting meaningful insights and enabling real-time decision-making across various domains. Future research directions in PDS include exploring novel techniques to improve accuracy-efficiency trade-offs, developing more robust algorithms for handling evolving data streams, and integrating with emerging machine learning frameworks for a holistic approach to data analysis.

In this project, its implementation and use has been showcased to analyse a biodiversity dataset and gain insight into the ecosystems of Indian forests. It offered a time and space efficient approach for providing the required results.

REFERENCES

1. Diaz, S., Sette, Á. L., & Brondizio, E. S. (2019). Assessing nature's contributions to people: Selecting the indicators for the Sustainable Development Goals. Ecological Economics, 160, 366-385.
2. Cardinale, B. J., Duffy, J. E., Gonzalez, A., Hooper, D. U., & Perroni, P. (2012). Biodiversity loss and its impact on humanity. Nature, 486(7401), 59-67. https://www.nature.com/articles/nature11148
3. Pressey, R. L., James, R. M., & Possingham, H. P. (2004). A framework for defining goals and measuring progress in reserve selection. *Conservation Biology*, 18(6), 1647-1657. https://www.sciencedirect.com/science/article/pii/S0959475214000218
4. Lindenmayer, D. B., & Burgman, M. A. (2005). *Monitoring and adaptive management of wildlife populations*. Blackwell Publishing Ltd.
5. Cormode, G., & Muthukrishnan, S. (2005). An introduction to data stream algorithms. *SIAM Journal on Computing*, 35(1), 185-207. https://www.cs.mcgill.ca/~denis/notes09.pdf
6. Muthukrishnan, S. (2005). Data streams: Algorithms and applications. *Now Publishers Inc.* https://www.nowpublishers.com/article/DownloadSummary/TCS-002
7. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press. [ISBN 9780262046305]
8. Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (2001). Handbook of applied cryptography (5th ed.). CRC Press. [ISBN 9780849330585]
9. Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7), 422-426.
10. Kirsch, A., & Mitzenmacher, M. (2006). Less is more: Sampling strategies for memory-efficient Bloom filters. ACM Transactions on Database Systems (TODS), 31(1), 420-450.
11. Broder, A., & Mitzenmacher, M. (2004). Network applications of Bloom filters. Internet Mathematics, 1(4), 485-509.
12. Cormode, G., & Muthukrishnan, S. (2005). An improved data stream summary: The count-min sketch and its applications. Journal of Algorithms, 55(1), 163-180. http://dimacs.rutgers.edu/~graham/pubs/papers/cmsoft.pdf
13. He, X., Fang, H., Luo, L., & Polychronopoulos, V. (2016). A stable PUF-based key generation scheme for resource-constrained devices. In Proceedings of the 53rd Annual Design Automation Conference (pp. 1-6). https://www.mdpi.com/2079-9292/10/14/1691
14. Patil, V., & Sarma, A. D. (2007). Sketching streaming data with cumulative sums and reservoirs. In Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms (pp. 866-875). https://dash.harvard.edu/bitstream/1/13777006/1/xrds.pdf
15. Flajolet, P., Fusy, É., Gandouet, O. and Meunier, F., 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete mathematics & theoretical computer science*, (Proceedings).